deep se
dependable evolvable pervasive software engineering group

Functions overload

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO
DI MILANO

Functions with default parameters

```
void print(string s = "Hello world");

int main(){
     print("Ciao!!");
     print();
}

void print(string s){
     cout << s << endl;
}
```

We can set a default parameter in case of "print()"

Overloaded functions (or methods!)

- Functions (methods) that have the same name but **different parameter** lists and that appear in the same scope are **overloaded**

```
1. void print(const char *cp);
2. void print(const int ia[], size_t size);

int j[2] = {0,1};

print("Hello World");// calls print(const char*)   This will call 1.
print(j, 2);// calls print(const int*, size_t)   This will call 2.
```

Overloaded functions

- Overloaded functions must differ in the number or the type(s) of their parameters
- It is an error for two functions to differ only in terms of their return types

```
Record lookup(const Account&);
bool lookup(const Account&);// error: only the return
                              // type is different
```

Calling an overloaded function

- **Function matching** (also known as **overload resolution**) is the process by which a particular function call is associated with a specific function from a set of overloaded functions
- For any given call to an overloaded function, there are three possible outcomes:
  - the compiler finds exactly one function that is a **best match**
  - there is no function with parameters that match the arguments in the call. Error: **no match**
  - there is more than one function that matches and none of the matches is clearly best. Error: **ambiguous call**

Calling an overloaded function

```
~~void f();~~
void f(int);
~~void f(int, int);~~
void f(double, double = 3.14);

f(5.6);
```

→ this means that we can call this function either passing two doubles or passing only one double and the other will be 3.14

- Identify the set of overloaded functions considered for the call:
  - **candidate functions** (all the functions named "f")
- Selects from the set of candidate functions those functions that can be called with the arguments in the given call:
  - **viable functions** (functions that can actually match the code)

Calling an overloaded function

```
~~void f();~~
void f(int);
~~void f(int, int);~~
void f(double, double = 3.14);

f(5.6);
```

- Viable functions: a function must have the same number of parameters as there are arguments in the call, and the type of each argument must:
  - match
  - or be convertible to the type of its corresponding parameter

Calling an overloaded function

- `f(int)` is viable because a conversion exists that can convert the argument of type double to the parameter of type int

- `f(double, double)` is viable because a default argument is provided for the function second parameter and its first parameter is of type double, which exactly matches the type of the parameter

Finding the *best* match, if any!

- Finally look at each argument in the call and select the viable function (or functions) for which the corresponding parameter best matches the argument
  - the closer the types of the argument and parameter are to each other, the better the match

- `f(int)` requires to convert the argument from double to int
- `f(double, double)`, is an exact match for this argument

- An exact match is better than a match that requires a conversion

- We call `f(double, double)`!

Function matching with multiple parameters

- f(42, 2.56);

- The viable functions are `f(int, int)` and `f(double, double)`

] **NO ONE** is the best match!

- There is an overall best match if there is one and only one function for which:
  - the match for each argument is no worse than the match required by any other viable function
  - there is at least one argument for which the match is better than the match provided by any other viable function

- If after looking at each argument there is no single function that is preferable, then the call is in error (ambiguous call)

Function matching with multiple parameters

- Consider the first argument
  - `f(int, int)` is an exact match
  - `f(double, double)`: the int argument 42 must be converted to double
  - A match through a built-in conversion is "less good" than one that is exact
- Consider the second argument
  - `f(double, double)` is an exact match to the argument 2.56
  - `f(int, int)`: the double argument 2.56 must be converted from double to int
- The compiler will reject this call because it is ambiguous
- In well-designed systems, argument casts should not be necessary

Overloading and `const` parameters

- A parameter that has a top-level const is indistinguishable from one without a top-level const

```
Record lookup(Phone);
Record lookup(const Phone);  // redeclares
                                  // Record lookup(Phone)
```

PASSING BY **VALUE**:
from the point of view of the compiler these 2 are equal, and so it's not overloading.

Overloading and `const` parameters

- We can overload based on whether the parameter is a reference (or pointer) to the const or nonconst version of a given type

```
Record lookup(Account&);// function that takes a reference
                              // to Account
Record lookup(const Account&);// new function that takes
                                  // a const reference
```

PASSING BY **REFERENCE**:
in this case the compiler sees these two as two different functions (overloaded)

Overloading member functions

- As with nonmember functions, member functions may be overloaded
- The same function-matching process is used for calls to member functions as for nonmember functions

```
class Screen{
private:
      unsigned x, y;
      char content[40][80];
public:
      char get() const;
      char get(unsigned x, unsigned y) const;
};

Screen myscreen;
char ch = myscreen.get();// calls Screen::get()
ch = myscreen.get(0,0);  // calls Screen::get(unsigned, unsigned)
```

Overloading based on `const`

- We can overload a member function based on whether it is `const`

```
class C{
public:
      f() const;
      f();
}
```

In classes we can overload a method based on the fact that is const or is not

- The non-`const` version will not be viable for `const` objects; we can only call `const` member functions on a `const` object
- We can call either version on a non-`const` object, but the non-`const` version will be a better match

DEMO

WE CANNOT CALL A NON-CONST METHOD ON A CONST OBJECT !!

References

- Lippman Chapters 6, 7